苏 州 大 学 实 验 报 告

院、系	计算机学院	年级专业	软件工程	姓名	朱金涛	学号	2327406014
课程名称	操作系统课程实践					成绩	
指导教师	王红玲	同组实验者	无		实验日期	2025 年 11 月 4 日	

实 验 名 称 虚拟内存实验

一、 实验目的

- 理解操作系统中缺页中断的工作原理。
- 学会通过修改内核实现统计系统缺页次数的方法。
- 学会观察/proc 中有关虚拟内存的内容。
- 学会通过使用相关工具来统计一段时间内的缺页次数。

二、 实验内容

- 通过修改 Linux 内核中相关代码，统计系统缺页次数。
- 通过查看/proc/vmstat 的变化来统计一段时间的缺页次数。

三、 实验步骤和结果

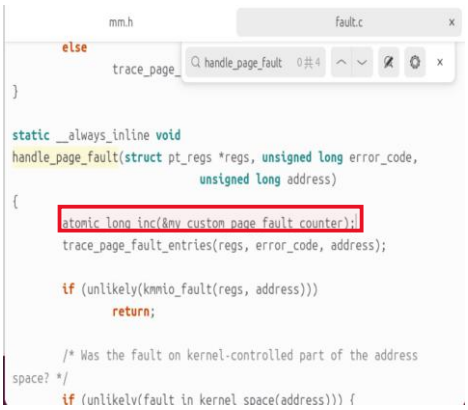
(一) 方案 1：通过修改内核代码，实现缺页统计。

1. 实验步骤：

首先在 arch/x86/mm/目录下找到 fault.c 文件，找到里面的 handle_page_fault()函数，这是**系统每次缺页都要调用**的函数。既然每次缺页都会调用，那么我们就可以就借此函数来统计缺页次数，只需两步：

(1) 声明变量：

(2) 在函数中加入自增逻辑:

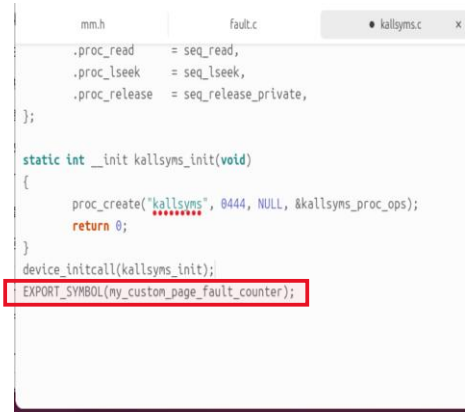


由此一来，就已经成功实现了记录缺页次数的功能。接下来要想办法把这个变量 my_custom_page_fault_count 能让内核中的.c 接收到:

(3) 通过在 mm.h 文件中添加 extern 声明

mm.h 就像是一个“公示栏”，内核中的其他.c 文件能够通过它来找到变量或函数的位置以及值。

最后一步通过 kallsyms.c 文件将缺页次数这一变量**导出（发布地址）**给内核之外的模块使用：



所有三个文件修改完毕后就进行内核的重新编译、安装、重启。

（4） 相关命令行：

sudo make -j32、sudo make modules_install、sudo make install、sudo reboot。

2. 实验结果：

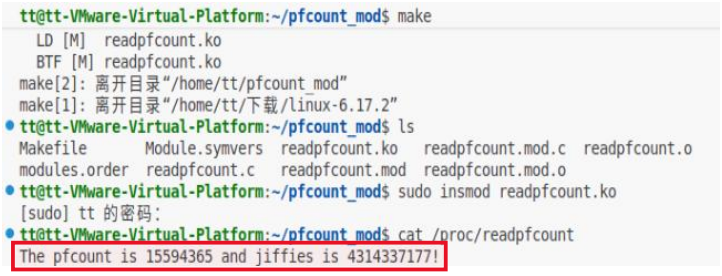
为了在用户空间获取内核变量，我编写了 readpfcount.c 这一内核模块，其核心原理是通过 extern 关键字向内核“请求”访问一个外部变量，并创建一个 /proc 虚拟文件来将这个变量的值显示给用户。我在模块代码中使用了 extern atomic_long_t my_custom_page_fault_counter; 这一声明，它充当一个**“请求”**，告诉内核加载器（insmod）我需要链接到由内核主程序（fault.c）通过 EXPORT_SYMBOL 导出的同名变量。当模块被加载时，我的 readpfcount_init 函数会执行，它调用 proc_create 来创建 /proc/readpfcount 文件，并将该文件的操作（通过 my_fops 结构体）最终链接到我的 my_proc_show 函数。因此，每当用户 cat 这个 /proc 文件时，my_proc_show 函数就会被触发，它会通过 atomic_long_read 安全地读取 my_custom_page_fault_counter 变量（此时已链接到其真实内存地址），并使用 seq_printf 将计数值和 jiffies（系统节拍数）打印出来。最后，我的

readpfcount_exit 函数也会在 rmmod 卸载模块时，调用 remove_proc_entry 来干净地移除这个 /proc 文件。

源码如下：

```c
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>
#include <linux/jiffies.h>
#include <linux/atomic.h>
/*
 * 声明我们在 fault.c 中导出的那个原子计数器。
 * 名字必须完全匹配 EXPORT_SYMBOL 中的名字。
 */
extern atomic_long_t my_custom_page_fault_counter;

/**
 * @brief 当 /proc/readpfcount 被读取时，此函数被调用。
 * (对应 PPT 中的 "my_proc_show") [cite: 49]
 */
static int my_proc_show(struct seq_file *m, void *v)
{
    /*
     * 使用 atomic_long_read() 来安全地读取原子变量的值。
     */
    seq_printf(m, "The pfcount is %ld and jiffies is %ld!\n",
               atomic_long_read(&my_custom_page_fault_counter),
               jiffies);
    return 0;
}

/**
 * @brief 当 /proc/readpfcount 被 "open" 时，内核调用此函数。
 */
static int my_proc_open(struct inode *inode, struct file *file)
{
    return single_open(file, my_proc_show, NULL);
}

/*
 * 将 /proc 文件操作（如 "read"）与我们的函数关联起来
 */
static const struct proc_ops my_fops = {
    .proc_open = my_proc_open,
    .proc_read = seq_read,
    .proc_lseek = seq_lseek,
    .proc_release = single_release,
};

/**
 * @brief 模块加载 (insmod) 时调用的初始化函数
 */
static int __init readpfcount_init(void)
{
    /* * 创建 /proc/readpfcount 文件 [cite: 53]。
     * 权限 0644 (所有者可读写，其他人只读)
     */
    proc_create("readpfcount", 0644, NULL, &my_fops);
    printk(KERN_INFO "/proc/readpfcount created\n"); // 在内核日志中打印一条消息
    return 0;
}

/**
 * @brief 模块卸载 (rmmod) 时调用的清理函数
 */
static void __exit readpfcount_exit(void)
{
    remove_proc_entry("readpfcount", NULL);
    printk(KERN_INFO "/proc/readpfcount removed\n"); // 在内核日志中打印一条消息
}

module_init(readpfcount_init);
module_exit(readpfcount_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Anders");
MODULE_DESCRIPTION("Module to read the custom page fault counter");
```

运行结果图如下：



表示当前的缺页次数为：15594365

(二) 方案 2：通过查看/proc/vmstat 信息获取缺页数。

1. 实验步骤：

编写 Python 程序实现查看：

(1) 核心函数：get_pgfault_count()

这个函数是脚本的核心，它负责从 /proc/vmstat 文件中精确地提取出 pgfault 的值。

- 打开文件：脚本首先会打开 /proc/vmstat 这个虚拟文件 。
- 逐行读取：它会一行一行地读取文件内容。
- 查找字段：它会持续查找，直到找到以 'pgfault ' 开头的那一行 。
- 解析数值：找到该行后（例如 pgfault 1234567），它会将这行文字按空格拆分，并提取出第二个元素（即那个数字 1234567）。
- 返回整数：最后，它将这个数字字符串转换为一个整数并返回。

源码：

```
def get_pgfault_count():
"""
```

```
打开 /proc/vmstat 文件，读取并解析 'pgfault' 的值。
"""
try:
# 打开 /proc/vmstat 文件
with open('/proc/vmstat', 'r') as f:
lines = f.readlines()

# 遍历文件的每一行
for line in lines:
# 寻找以 'pgfault ' (注意有个空格) 开头的行
if line.startswith('pgfault '):
# 按空格分割，获取第二个值（即数字）
parts = line.split()
if len(parts) >= 2:
return int(parts[1])

# 如果循环结束了都没找到
print("错误：未能在 /proc/vmstat 中找到 'pgfault' 字段。")
return None

except FileNotFoundError:
print("错误：/proc/vmstat 文件不存在。请确保在 Linux 系统上运行。")
return None
except Exception as e:
print(f"读取文件时发生错误: {e}")
```

```
return None
```

主程序逻辑就是：在程序开始时先读取一次 /proc/vmstat 中的 pgfault 值（获取初始值），接着暂停等待一段设定的时间（5 秒），时间到了之后再读取一次 pgfault 的值（获取结束值），最后用这个结束值减去初始值，得到的差值就是这段时间内系统新发生的缺页次数。

2. 实验结果：

由上图可见，程序能够分别读取相隔 5 秒的缺页次数，并计算其差值。

四、 实验总结

1. 通过对比两种方法统计结果来验证实验的准确性：

手动计算方法一的缺页差值：30055739 - 29984089 = 71650

（由于对于方法一我是手动命令行操作，不能完全保证与方法二同时进行，所以略有偏差）

但是从提看来，二者缺页次数的数量级一致切差异很小很小，可以认为二者测量结果一致，实验成功！

2. 思考题:

(1) 说明该实验中统计缺页次数的原理，并说明其合理性。

答: 本实验统计缺页次数的原理有两种: 一是通过修改内核源代码（fault.c ），在处理缺页中断的核心函数（handle_mm_fault）中植入一个全局计数器 ，使每次中断都直接触发该计数器递增 ；二是通过读取 /proc/vmstat 文件中的 pgfault 字段，该字段是系统自启动以来的累计缺页总数,通过计算其在一段时间内的差值来得到该时段的新增缺页次数 。这两种方法都是合理的: 方法一直接在中断源头插桩 ，统计结果最直接; 方法二则是读取内核自身维护的官方统计值 ， 同样准确可靠。

(2) 如何验证实验结果的准确性?

答: 验证实验结果准确性的最佳方法是**交叉对比**。我们可以同时使用实验中的两种方法进行测量: 首先在 T1 时刻同时记录方法一(cat /proc/readpfcount)的计数值 V1 和方法二（/proc/vmstat 中的 pgfault ）的计数值 V1'; 然后在系统运行一段时间（并产生一些缺页负载 ）后，在 T2 时刻再次同时记录 V2 和 V2'。通过比较两个增量（V2 - V1）和（V2' - V1'），如果两者几乎完全相等，就证明了我们修改内核（方法一）和读取 /proc（方法二）的实验结果都是准确的。

(3) 总结说明一下在/proc/下增加一个项的基本过程。

答: 在 /proc 下增加一个项的基本过程（如实验模块所示）: 首先定义一个 proc_ops 结构体,并将其中的 proc_open 操作链接到一个自定义的“open”函数;这个“open”函数再通过 single_open 辅助函数，将“读”操作绑定到一个 my_proc_show 函数 。接着，在 my_proc_show 函数中，我们使用 seq_printf 来将内核中的变量（如我们的

缺页计数）写入文件序列。最后，在模块的 init 函数中调用 proc_create()，传入新项的文件名（如 "readpfcount"）、文件权限和 proc_ops 结构体 来完成注册；并在 exit 函数中调用 remove_proc_entry 将其移除。

(4) 总结实验过程中出现的问题及解决方案。

答：实验中最大的问题是，实验指导 PPT 中的代码（如 do_page_fault 函数 和 pfcount++ ）与我使用的现代 Linux 内核版本不兼容。**解决方案是**：我通过分析新版 fault.c 源码，找到了正确的插桩函数 handle_page_fault；并将 unsigned long 计数器 升级为线程安全的 atomic_long_t，使用 atomic_long_inc() 进行递增。另一个问题是 EXPORT_SYMBOL 放在 kallsyms.c 中导致编译时出现“未定义引用”错误，**解决方案是**在 mm.h 中做好声明。

3. 实验小结：

在此次操作系统实验中，我围绕虚拟内存管理和缺页中断机制，成功地通过两种截然不同的方法实现了对系统缺页次数的统计，并完成了交叉验证。

在方法一（修改内核）中，我遇到了第一个主要挑战：实验指导 PPT 中的函数（如 do_page_fault）与我实际使用的现代内核版本不符。通过分析 fault.c 源码，我定位到正确的插桩点为 handle_page_fault 函数。同时，考虑到多核环境下的线程安全，我将 PPT 中的 unsigned long 计数器升级为 atomic_long_t，并使用 atomic_long_inc() 进行递增 。在编译内核时，我遇到了“未定义引用”的错误，通过在 mm.h 中添加 extern 声明，并在 kallsyms.c 中导出符号，我成功解决了链接问题。内核编译重启后，我编写了 readpfcount.c 模块，通过 extern 关键字请求内核导出的计数值，并调用 proc_create

创建了 /proc/readpfcount 接口 。当我 cat 此文件时，my_proc_show 函数被触发，通过 atomic_long_read 成功读取并显示了内核中的计数值。

在方法二（读取/proc/vmstat）中，我编写了一个 Python 脚本 。该脚本的原理是读取 /proc/vmstat 文件，解析出 pgfault 字段的值。我的主程序逻辑是：在 T1 时刻读取一次 pgfault 值，等待 5 秒钟，在 T2 时刻再次读取，最后计算两次读数的差值。实验结果表明，该脚本能准确计算出指定时间段内的缺页次数增量。

最后，我进行了**交叉验证**。我分别用方法一和方法二在相近的时间段内统计缺页增量，方法一（手动 cat）的结果为 71650 次，方法二（Python 脚本）的结果为 71649 次。尽管由于手动操作存在微小的时间偏差，但两个结果在数量级上完全一致，差异极小，有力地证明了两种实验方法均已成功实现且结果准确可靠。本次实验不仅让我深刻理解了缺页中断的工作原理，更锻炼了我分析内核源码、解决版本兼容性问题以及编写内核模块（包括 proc 文件创建、符号导入导出）的实践能力。